TWINE RISC : ARCHITECTURE AND PERFORMANCE EVALUATION STUDY

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the degree of

MASTER OF TECHNOLOGY

by

Dhiren Patel

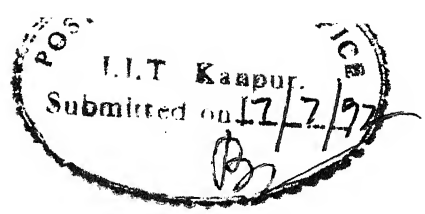
to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JULY 1992

CERTIFICATE



It is certified that the work contained in the thesis entitled TWINE RISC : ARCHITECTURE AND PERFORMANCE EVALUATION STUDY, by DHIREN PATEL (Roll No : 9021101) has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(Dr. Rajat Moona)
Deptt. of CSE
IIT Kanpur.

SE- 1992-M-PAT-TWI

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to Dr. Rajat Moona, my thesis supervisor for his expert guidance and constant encouragement through out the course of this thesis work. Working with him, in the very cordial atmosphere he created, has been a special and memorable experience to me.

I am thankful to all my friends, especially Mr. T. Agrawal, Mr. A. Singhai, Mr. H. Parekh, Mr. D. Gupta, and Mr. Dinesh Rao, who helped me in programming and gave valuable suggestions.

I gratefully acknowledge the sponsorship provided by my institute S.V.Regional College of Engg. & Tech., Surat, Gujarat, for M.Tech. study.

Finally I would like to thank Hall IV residents who made my stay at IITK an enjoyable and memorable one.

ABSTRACT

As device technology is approaching fundamental limits, future increases in computing power with improved cost/performance ratio will be forced to rely on advances in computer architecture.

Twine RISC is a novel single chip low cost processor architecture which exploits instruction level temporal parallelism by its well engineered RISC pipeline and spatial parallelism by allowing multiple threads of computation to coexist and execute in parallel. In this project, a simulator for evaluating Twine RISC is developed. Key issues involved in design of architecture are generation and synchronization of threads and support for split phase transactions for data transfer to/from global shared memory. Technological constraints such as available VLSI technology(which decides chip size) and state of the art memory technology are also considered.

Table of Contents

List of Figures

Chapter 1 : Introduction and Thesis Organization

1.1 Introduction :

Fine grain multicomputers offer the potential of a signi-icant increase in maximum computing power with greatly improved ost/performance ratio. Significant challenges exist in processor rchitecture for these machines.

RISC processors exploit instruction level arallelism(Temporal parallelism) whereby an instruction pipeline s kept busy and performs more than one operations for various instructions. A significant amount of easily detectable parallelism actually exists in most general purpose codes. Dataflow architectures appear to be the most suitable for exploiting such parallelism as they support generation and coordination of parallel activities directly in hardware and can tolerate long unpredictable communication delays [3]. There has been a consistent convergence toward a "practical" architectural framework for implementing dataflow machines. The dataflow/von Neumann hybrid architecture is a new phase of evolution in computer architecture to exploit both temporal as well as spatial parallelism [2].

The context of this thesis work is to simulate a novel processor architecture called Twine RISC and to enhance it. Twine RISC is a low cost single chip processor architecture which exploits instruction level parallelism by its well engineered RISC pipeline and spatial parallelism by allowing multiple

threads of computation to coexist and execute in parallel.

1.2 Thesis Organization :

The rest of the thesis is organized as follows : In Chapter 2, we provide background and related work toward "practical" architectural framework for dataflow/von Neumann hybrid architectures. In chapter 3, architecture of the Twine RISC processor is discussed. Chapter 4 deals with the software environment for the Twine RISC. In chapter 5, simulator to test and evaluate Twine RISC is discussed. Finally chapter 6 is the concluding chapter of the thesis. It contains a brief summary and discussion on future work in this area.

Appendix A gives complete instruction set and execution flow policy for Twine RISC. Appendix B is a condensed specification of the simulator. Appendix C describes Input/Output specifications and User's manual for simulator. Some test programs and performance results are also included.

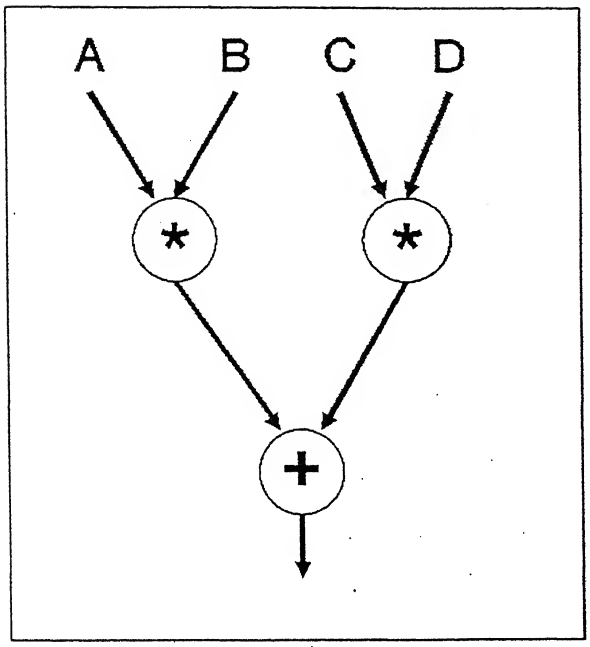


Fig 2-1.1 Dataflow graph for expression (A*B) + (C*D)

Chapter 2. Background and related work

2.1 Introduction and Overview :

In this chapter we discuss in short the dataflow graphs and their ability to represent maximum available parallelism in a program. There have been several attempts of building machines capable of executing dataflow graphs. We discuss some of them in Section 2.2. In Section 2.3 dataflow/von Neumann hybrid architectures are discussed. Finally in section 2.4 we discuss enhancements in conventional RISC architecture for providing support for multithreading. These include enhanced memory model, split phase transactions and primitives for multithreading.

2.1.2 Dataflow graphs :

Dataflow graphs are powerful intermediate representations for compilers. They are directed graphs in which nodes represent primitive functions such as ADD, SUB .. etc., and the arcs represent data dependencies between functions. Dataflow graphs specify only a partial order for the execution of instructions and thus provide opportunities for parallel and pipelined execution at the level of individual instructions. For example, the dataflow graph for the expression [a*b + c*d] only specifies that both multiplications be executed before the addition, however, the multiplications can be executed in any order or even in parallel. The advantage of this flexibility becomes apparent when we consider that the order in which a, b, c, and d will become available may not be known at compile time.

This strategy implicitly introduces sequencing between instructions which depend on each other, but allows instructions to execute in parallel if there exist no dependency between them. So it is very clear that if dataflow graphs are executed directly, the machine can exploit maximum available parallelism in computation [3,10].

2.2 Dataflow architectures :

Dataflow architectures are language based architectures in which dataflow program graphs are the base language. Here dataflow graphs constitute a formal interface between dataflow architectres and user programming languages.

We can view dataflow graphs as a machine language for a parallel machine where a node in a dataflow graph represents a machine instruction. Each instruction contains an opcode and a list of destination instruction addresses.

An instruction or a node may execute whenever token(operand data) is available on each of its input arcs and that when it fires(i.e. operation is performed on its input tokens), the input tokens are consumed, a result is computed, and a result token is produced on each output arc, which may be an input token for another node in the graph.

This dictates the following basic instruction cycle :

a. Detect when an operation is enabled(when all operands values available)

b. Determine the operation to be performed, i.e. fetch instruction.

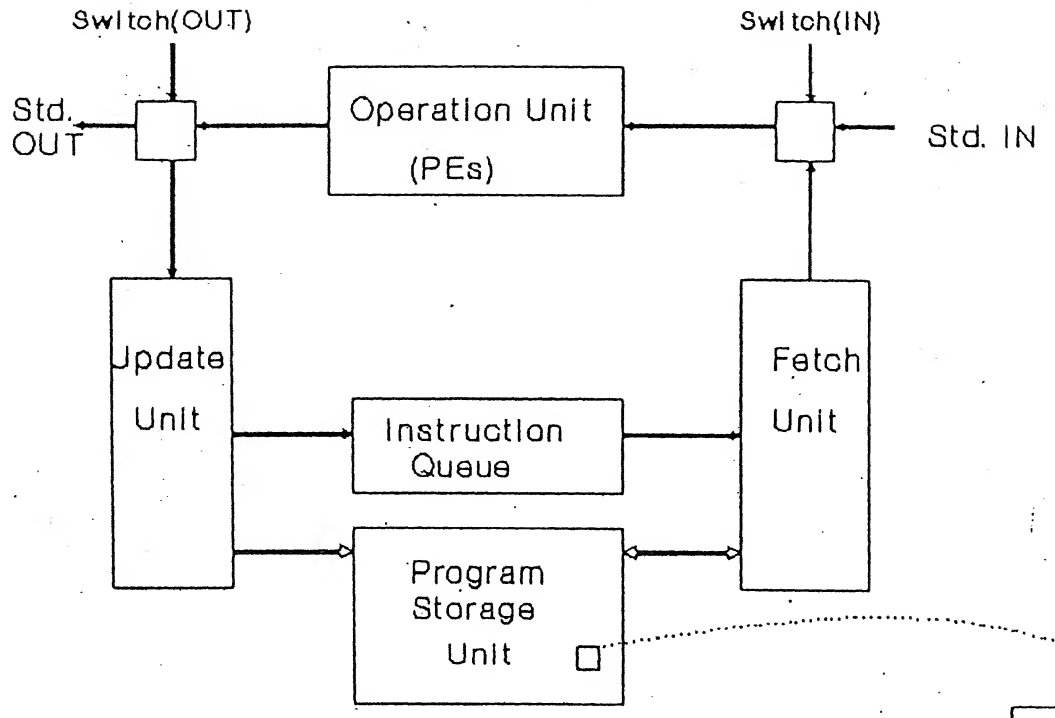


Fig 2.2-1. Static Dataflow Architecture

Opcode
Operand Left + Flag
Operand Right + Flag
Acknowledgment Counter
Destination #1
Destination #N

An Activity Template

c. Compute results

d. generate result tokens

This is the basic instruction cycle of any dataflow machine, however, there remains tremendous flexibility on the details of how this cycle is performed [3].

There has been a cosistent convergence toward a "practical" architectural framework for implementing dataflow machines. Several architectures on dataflow concept have been proposed, some of which have been implemented in experimental machines [2].

Examples are :

1. Static Dataflow Machine Projects :

- The MIT Static Dataflow Machine [3]
- The NEC Dataflow Machines NEDIPS and IPP [3]

In these machines, data tokens are assumed to move along the arcs of the dataflow program graph to the operator nodes. The nodel operation gets executed when all its operand data are present at the input arcs. Also all output arcs of a node be empty before that node is enabled. A token moves to the next unit only after that unit has signalled that it can accept the token. Only one token is allowed to exist on any arc at given time. The restriction cannot be enforced at hardware level, but its effect can be achieved by executing only graphs that have the property whereby no more than one token can reside on any arc at any stage of execution.

Basic model of static dataflow machine architecture is shown in fig.# 2.2-1.

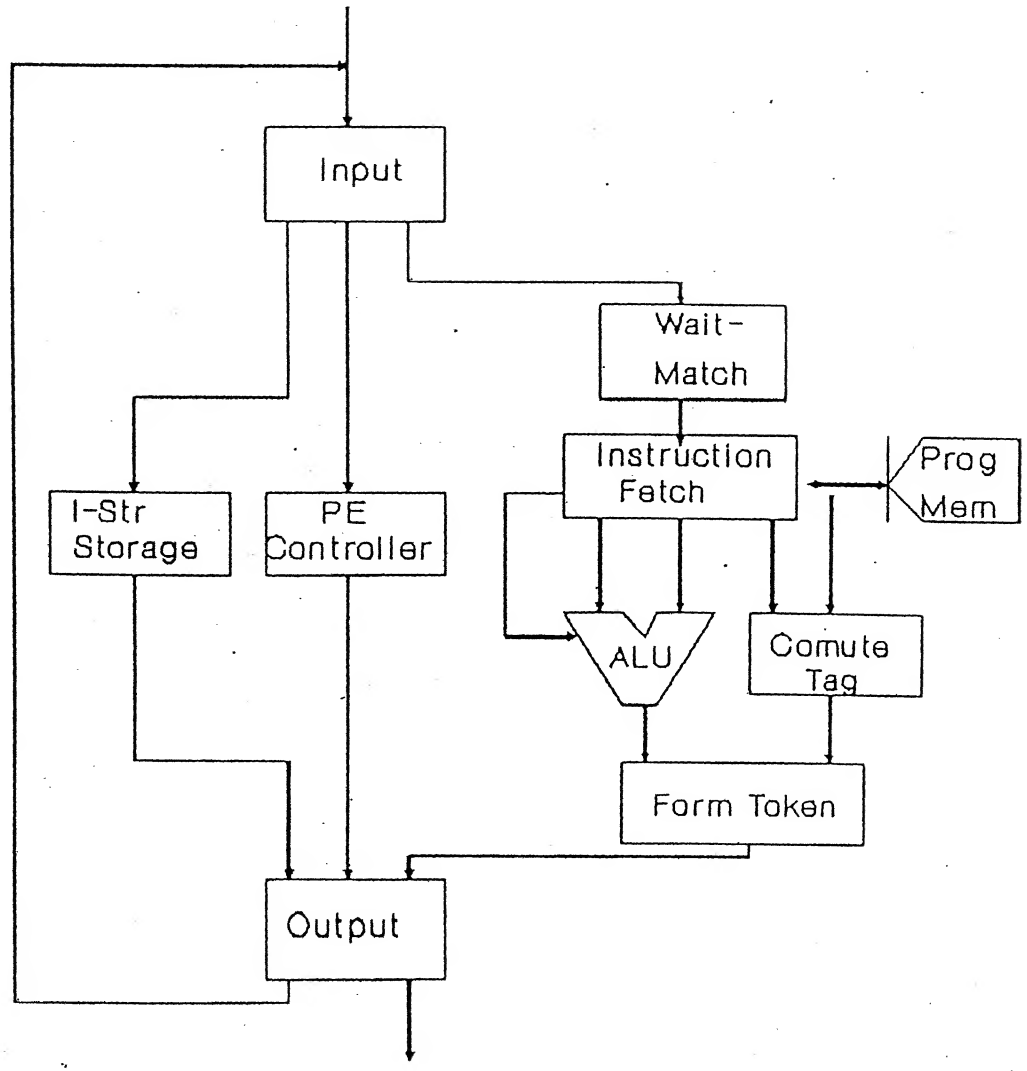


Fig 2.2-2. PE for Tagged Token Dataflow Machine

. Dynamic (Tagged Token) Dataflow Machine Projects :

- The Manchester Dataflow Machine [17]
- SIGMA 1 at Electrotechnical Laboratory, Japan [18]
- The MIT Tagged Token Machine [5,7]
- Monsoon : an Explicit Token-Store Architecture(MIT)),14]

These machines use tagged tokens, so that more than one oken can exist on an arc. The tagging is achieved by attaching a abel with each token which uniquely identifies the context of hat token. A node is identified by a pair, code block and nstruction address. Tags have four parts, viz; invocation ID, teration ID, code block, and instruction address. The iteration ID distinguishes between different iterations of a particular invocation of a loop code block, while the invocation ID distinguishes between different invocations. If the graph is cyclic, the tagging allows dynamic unfolding of the iterative computations and thereby exploits maximum available parallelism.

Basic processing element architecture model of the tagged token machine is shown in fig.#2.2-2.

Some defects of these machines are as follows:

1. A circular pipeline does not work well as a "pipeline" for less parallel execution. It may occur that only one token is going round the pipeline cycle, and that PE throughput is less than one per a pipeline circular time.

2. Simple packet-based architecture cannot exploit registers or a register file efficiently. As token is always realized as a

packet and each of the packets enter a PE whenever possible, it is nonsence to reserve tokens in registers for the future node operation. This is one of the main reason why a fine pitch pipeline is difficult to implement in a dataflow machine.

3. For matching hardware and time complexity are heavy.

4. Packet flow traffic is too heavy.

5. It takes much time to eliminate garbage tokens, which are generated while executing switch operations for conditional computations.

2.3 Dataflow/von Neumann Hybrid Architectures :

In the previous section we have reported several shortcomings of pure dataflow machines. It has been realized that to overcome these shortcomings following changes in the design are necessary [11,13,16].

1. Improve machine performance by integrating a packet based circular pipeline of dataflow machines and a register based advanced control pipeline of von Neumann machines.

2. Use RISC based single chip PE design to simplify architecture, and a direct matching scheme with large register file.

These lead to development of Dataflow/von Neumann hybrid architectures which can exploit both conventional von Neumann and dataflow compiling technology.

Examples of srchitectures falling in this class are P-RISC(MIT)[13] and EMC-R(Electrotechnical Laboratory, Japan) [16].

P-RISC :

P-RISC(for Parallel RISC) can be viewied as a dataflow chine that can achieve software compatibility with conventional n Neumann machine. Distinctive features are RISC like 3-address structions that opoerate entirely within a processing element, AD/STORE instructions to move data in and out of the PE, I-ructure type storage model. Collection of frames on a PE is garded as a collection of register sets, a particular register t being identified by frame pointer. FORK and JOIN are instrucons for thread initiation and synchronization. The processor pe and the token queue form a ring around which tokens are cir-ılated[13].

EMC-R :

MC-R is a PE for a parallel computer EM-4 built at Electrotechncal Laboratory, Japan. Distinctive features of EMC-R architecure are strongly connected arc dataflow model, a direct matchng scheme and register based sequencing, a RISC based design, nd an integration of a packet based circular pipeline and a egister based advanced control pipeline[16].

2.4 Enhancement and Support towards Multithreading :

Two fundamental issues in multiprocessing/multithreading are memory latency and waits for synchronization events. Both are very expensive on von Neumann machines[4][6].

Memory latency :

Memory latency is defined as the time which elapses between making a request and receiving the associated response from memory. In a von Neumann processor, memory latency determines the time to execute memory reference instruction, which finally determines the maximum instruction processing speed. Most von Neumann processors are likely to be "idle" during long memory references, and such references are unavoidable. In order to reduce memory latency cost, it is essential that a processor be capable of issuing multiple overlapped memory requests[7].

A different memory model, I - structure, is used in a few dataflow machines to tolerate memory latency thereby incresing overall throughput. The transactions for processor to I-structure memory are in terms of messages and are termed as split-phase transactions.

I - structure Memory[2] :

The basic idea behind I-structure storage is to defer a data-read if the corresponding location has not been written.

Here each storage cell contains status bits to indicate that the cell is in one of three possible states.

a. EMPTY : Nothing has been written into the cell since it was last allocated. No attempt has been made to read the cell. It may be written as for conventional memory.

b. FULL : The cell contains valid data that can be freely read as in a conventional memory. In a conventional I-structure memory, any attempt to write a FULL cell is signalled as an error. However, in our model FULL cell can be overwritten.

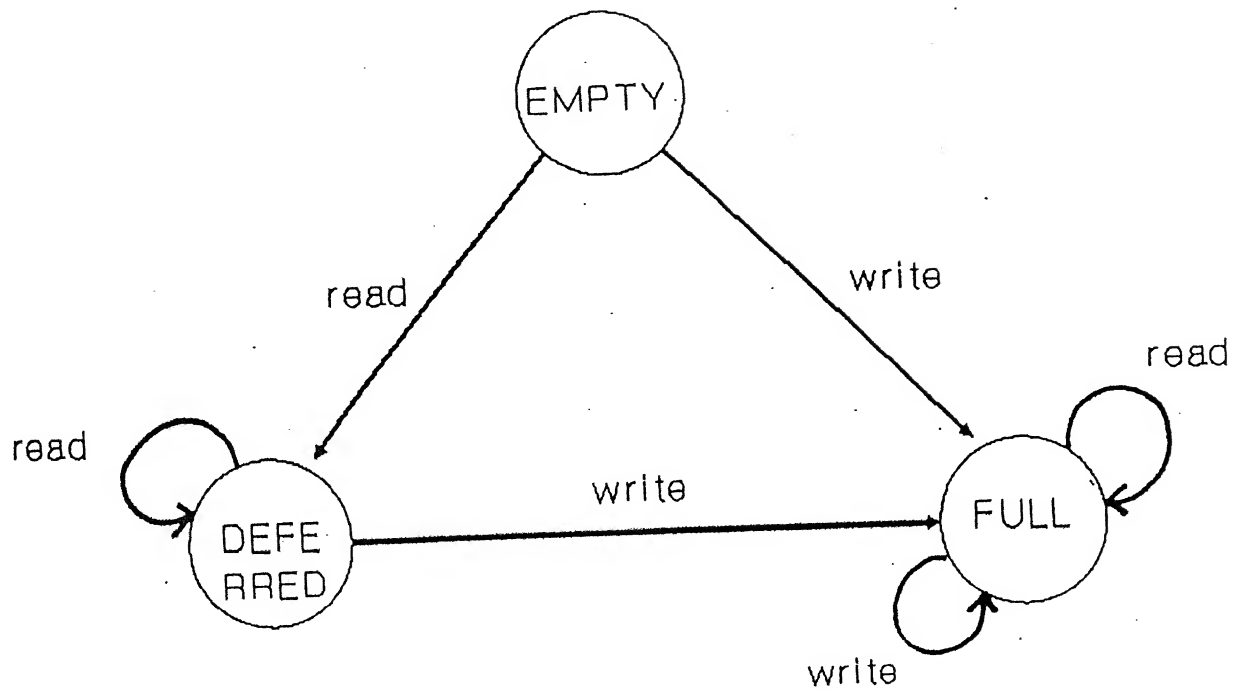


Fig 2.4-1. State Transition Diagram for an I-structure Cell

c. DEFERRED : Nothing has been written into cell, but at least one attempt has been made to read it. When it is written, all deferred reads must be satisfied.

Cells change state in the obvious ways when presented with requests.

In fact, reads and writes may even get out of order in communication networks. Synchronization of reads and writes is performed for each cell of the structure, so that there is no problem when a read preceds the write to a cell. In such a case, a deferred read state is created whereby the read is put aside on a list with a promise to fulfill the read request when the cell is written. This synchronization is implemented using two bits to indicate whether a cell is in empty, full or deferred state. A pointer to the deffered read list, if one exists, is kept in the empty cell of the structure until the point when the expected write takes place.(fig.# 2.4-1)

Split Phase Transactions :

It is a method by which synchrony between the I - structure request and reply is maintained. A request token is sent to I - structure unit(potentially across the communication network) and the processor is then free to continue executing other instructions while the request is being delivered and handled. This will never cause the processor pipeline to stall.

Threads :

Threads are defined as small processes that operate

almost entirely on local data and rarely interact. They are the basic blocks/units of computation in to which programs are decomposed for parallel execution. Threads can be created dynamically during computation and die after having produced and consumed data. Threads can be in one of three states : ready to execute(queued locally or globally), executing, suspended(waiting for synchronization signal).

Following modifications are required to a conventional RISC to make it suitable for exploiting the fine grain parallelism of dataflow execution, while still retaining the effecient control mechanism of von Neumann computing[15,16].

1. Modify the RISC processor implementation to make it multithreading.

- Implement more than one PEs on a single chip.
- Include primitives MFORK and MJOIN for generating and synchronizing multiple threads of computation.

2. Augment the RISC processor with I - structure like storage with split phase transactions.

The Twine RISC architecture implants the following features.

1. Loads(memory request) are split phase transactions. Therefore, responses can come back in any order.
2. The processor switches automatically to another thread of computation if it exists rather than being idle.
3. The processor supports multiple threads.
4. The pipeline is kept full as long as token queue is not

empty.

5. Simultaneous execution of various threads is possible and is carried out within the processor.

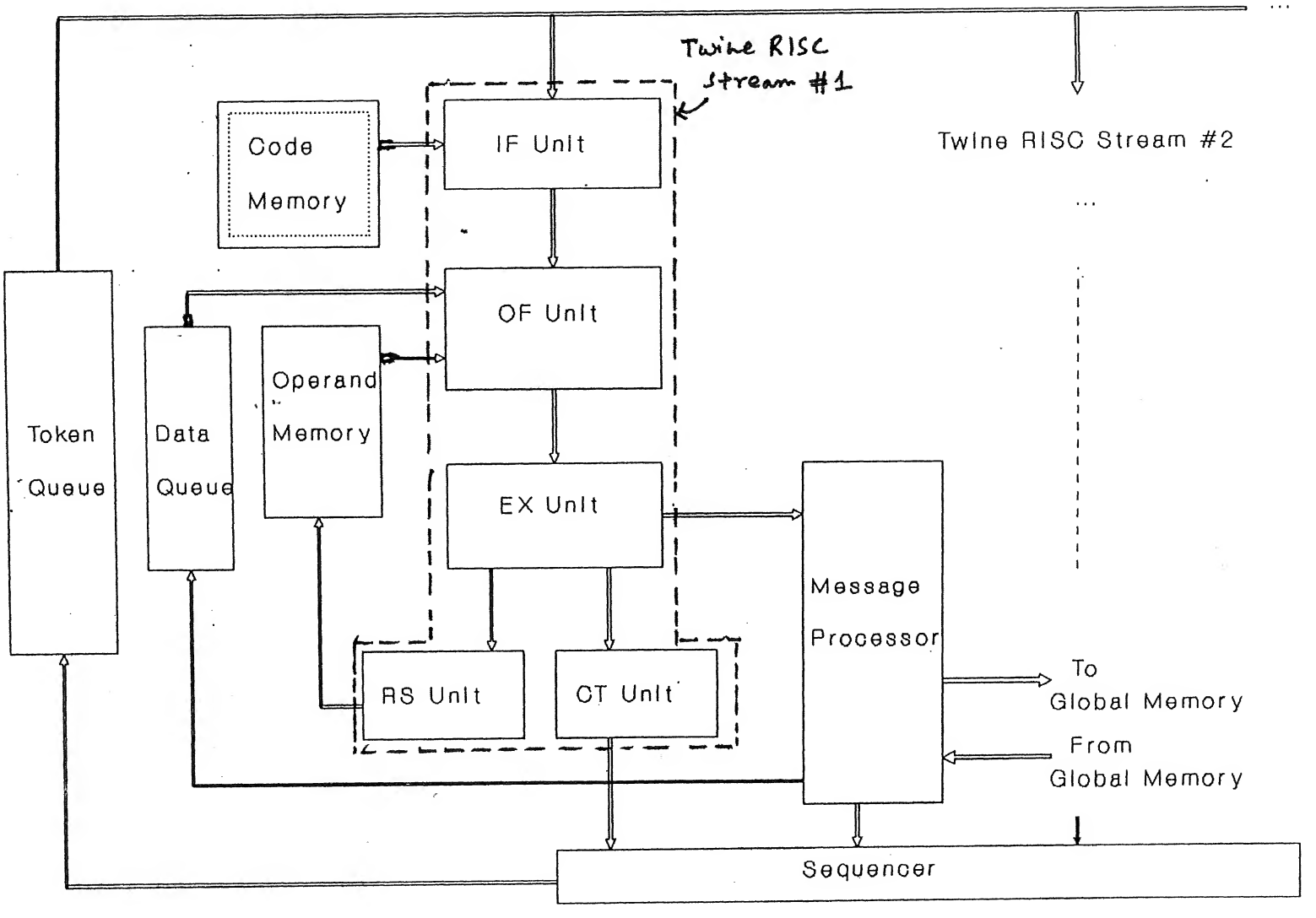


Fig.3-1. Twine RISC Processor Architecture

Chapter 3 : Twine RISC : Its Architecture

3.1 Introduction and Overview

In this chapter we discuss complete processor architecture of the Twine RISC. We adopt a RISC architecture for the Twine RISC for its simplicity and execution efficiency. Instruction level parallelism is exploited in the context of sequential thread executing in a well engineered RISC pipeline. Multithreading is exploited by providing more than one streams of execution pipeline on single chip. These streams are called Twine RISC Streams(TRS). In Section 3.2 we discuss various blocks of Twine RISC viz., Code Memory, Operand Memory, Token Queue, Sequencer, Data Queue, and Message Processor. The Twine RISC processor also supports split phase transactions between memory and processor through Message Processor and Data Queue. We also discuss this mechanism in Section 3.2. Instruction pipeline in a TRS is discussed in Section 3.3. Various stages in this pipeline include Instruction Fetch Unit, Operand Fetch Unit, Execution Unit, Result Store Unit and Continuation token generation Unit. (see fig 3.1)

3.2 Various Building Blocks

3.2.1 Code Memory (CM) :

Code memory holds instructions. Each TRS has an access to a CM outside the chip. These CMs are read only memories for TRSs. A separate host processor is used to initialize the CM by loading a Twine RISC progarm chunck.

3.2.2 Operand Memory (OM) :

It is a register file of 64 registers each 32 bits wide. Operand memory is shared by all TRSs. All TRSs can simultaneously write to this OM at different locations and read operands from it simultaneously. As clear by the instruction pipeline, each TRS has requirements of 2 reads and 1 write per cycle. As the Twine RISC processor can have more than one TRS to support spatial parallelism, the Operand Memory has 2N read ports and N write ports for N TRSs.

3.2.3 Token Queue (TQ) :

Token Queue feeds TRSs with the continuation tokens. A continuation token is formed with two pointers, viz; a frame pointer(FP) and an instruction pointer(IP). IP indicates the location of instruction to be executed in the Code Memory. While FP is a base pointer to the set of operands in data memory(OM) analogous to the base address of an activation frame for a procedure invocation. By using frame relative addressing the same code block can have multiple active invocations. Since the continuation tokens generated by any of the TRSs correspond only to the start address of different threads, they can be picked up by any other TRS in the Twine RISC processor.

3.2.4 Sequencer :

The continuation tokens generated in the system are stored in the TQ through a Sequencer. The Sequencer samples continuation tokens generated by various TRSs and stores them in TQ.

As these tokens generated by TRSs are independent of each other, the sequence in which they are stored in the TQ is irrelevant and a program works independent of any sequencing scheme forced by the system. This makes the design of the Sequencer relatively simple.

3.2.5 Data Queue (DQ) :

It is an alternate operand memory for special instruction RESM. RESM does not refer OM, instead it reads data from DQ and treats them as operands. DQ is an inevitable hardware which enables OM to be loaded. When a memory operation LOAD/LOADX is issued, upon completion of the operation a message is returned to the Message Processor by the external memory controller. This message contains a value, continuation token and destination register. These data are written in the DQ. When RESM instruction is executed data is finally moved from DQ to OM and the thread reinitiates.

3.2.6 Message Processor (MP) :

MP handles message traffic between the processor and external memory. It also implants split - phase transactions where the requests for read/write to global I-memory are dispatched from all TRSs through MP. The MP receives read/write requests from various TRSs in the processor and forwards them to the external interface for global I-structure memory controller. In case of a read request, the MP eventually receives a message from the external interface containing value, operand memory

location and continuation token. Upon receiving such a message, the MP writes data into DQ and generates a continuation token <FP.0>. This is essential as the MP can't store data in OM. Corresponding instruction RESM(at location 0 in CM) takes this data from DQ and stores it in OM, besides generates the continuation token.

3.3 The TRS Pipeline :

The TRSs in a Twine RISC processor essentially capture spatial parallelism between different threads of computation. Within a TRS the various stages are instruction fetch unit(IFU), operand fetch unit(OFU), execution unit(EXU), result store unit(RSU) and continuation token unit(CTU). All these units operate asynchronously with hand shake signals. There is a buffer between two successive units.

3.3.1 Instruction Fetch Unit (IFU) :

Initially it fetches new token address from TQ and fetches instruction from CM. It determines whether the next instruction to be fetched from subsequent location or not. For instance in the case of various arithmetic/logic instructions, the next instruction comes from subsequent location. i.e IP -> IP + 1. However in case of branch instructions the location of next instruction is not determined at IF stage so IFU fetches a continuation token from the TQ and starts another thread.

Instruction set is organized in such a way that by looking at the first bit of opcode IFU can determine whether the next

instruction is to be fetched from IP + 1 or a new thread is to be started.

To prevent race between MJOIN instructions IFU also detects MJOIN instruction and stalls other TRS pipelines for MJOIN instructions by setting MJOIN lock line. This way, the MJOIN instruction is executed in atomic and exclusive manner. Opcode for MJOIN is chosen to be 111111 so that the detection hardware at IF Unit is simplified.

Finally IFU prepares a packet
<6 bit opcode,6 bit R1,6 bit R2,6 bit R3>
and sends it to OFU through buffer between IFU and OFU.

3.3.2 Operand Fetch Unit (OFU) :

This unit recognizes instructions partially by decoding 3 bits of opcode and decides the number of operands to be fetched from operand memory. This unit also decodes RESM instruction in which case it fetches operands from DQ. It then generates a packet
<6 bit opcode, 32 bit left operand, 32 bit right operand, 6 bit destination register>
which is sent to the EXU through its input buffer. Once the fetch is done, the handshake signal from OFU to IFU causes IFU to resume its operation.

3.3.3 Execution or Functional Unit (EXU) :

This unit is identical to conventional ALU except that it

generates continuation tokens for branch and other special instructions for thread instantiation. It prepares a packet

<32 bit result value, 6 bit destination register>

and forwards it through buffer to RSU. It also prepares token <FP.IP> for CTU and forwards it through buffer queue. For split - phase transactions(for memory read/write) EXU sends request message to MP and continue. It implements the handshake signal with OFU.

3.3.4 Result Store Unit (RSU) :

This is the only stage which can write in to shared OM. It writes the result value in destination register. It releases MJOIN line set by IFU in case of MJOIN instruction.

3.3.5 Continuation Token Unit (CTU) :

It forwards new thread token (FP.IP) to Sequencer.

Both RSU and CTU implement handshake signals with EXU.

Category	Instruction	Action
Airthmetic and Logic	ADD	Integer add
	SUB	Integer subtract
	AND	bitwise AND
	OR	bitwise OR
	XOR	bitwise XOR
	SFTL	shift left
	SFTR	shift right
Branch	JMP	direct jump
	JZ	jump on zero (equal to)
	JP	jump on positive (greater than)
	JPZ	jump on positive or zero (greater than or equal to)
	JNZ	jump on negative or zero (less than or equal to)
Data transfer from/to memory	LOAD	load from external memory to OM/register
	LOADX	" "
	RESM	move data from DQ to OM/register
	STORE	store to external memory from OM/register
	STOREX	" "
Thread gen. and synch.	MFORK	generation of multiple threads
	MJOIN	synchronization of multiple threads

Fig.4.2-1. Instruction Set

Chapter 4. Twine RISC : Its Software Environment

4.1 Introduction and Overview

In this chapter we discuss software support available in Twine RISC and details of its implementation. In section 4.2 we provide the insruction set of Twine RISC. Twine RISC supports multiple threads of execution. We discuss the support for creating and synchronizing multiple threads in Section 4.3. In Section 4.4 instructions that supports memory references between Twine RISC and outside shared global memory are discussed. Section 4.5 summarizes the software environment of Twine RISC.

4.2 Instruction Set and its Coding :

The instruction set of a Twine RISC is intended to be a simple extension of the RISC model. There are totally 19 different instructions, each of which is capable of being executed in a single clock cycle. Instructions are broadly classified into two classes viz; Ordinary RISC like instructions and special instructions. Special instructions are to support generation and synchronization of multiple threads and to handle external memory references as split phase transactions. With these special instructions it is possible to simulate the fine grained, asynchronous parallelism of dataflow execution.

The instruction set of Twine RISC is coded in a way that by decoding minimum number of bits, instructions are recognized at various units of TRS pipeline. (See Appendix A). For example

IFU decodes only one bit of instruction opcode to determine the next instruction fetch, whether it should be from code memory or from token queue.

The instructions available in Twine RISC are given in table#. Fig. 4-2-1.

4.3 Handling Multiple Threads

To implement concurrent threads Twine RISC supports instructions MFORK and MJOIN. MFORK creates multiple threads while MJOIN synchronizes them.

4.3.1 MFORK : Generation of multiple threads

MFORK instruction is a method of spawning parallel threads of computation from within an executing thread. By using this instructions upto a maximum of 5 threads are created. One of these threads is the parent thread with continuation <FP.IP + 1>. Addresses of new threads to be generated are kept in a 32 bit operand. Each address is relative to the address of MFORK instruction and is specified in 8 bits. Thus upto 4 addresses are stored in a single 32 bit operand. Execution of this instruction causes continuation token <FP.IP + byte offset> to be generated for each non zero vzlue of offset. The number of threads thus created is stored in a location which can later be used by MFORK's dual instruction MJOIN for synchronizing execution.

4.3.2 MJOIN : Synchronization of multiple threads

MJOIN instruction allows multiple threads to synchronize. Content of the OM location specified in MJOIN instruction is decremented by 1 for each execution of MJOIN. This location is set by MFORK instruction to the number of threads. As each thread calls MJOIN exactly once, the only thread which finds this location equal to zero after decrementing is the last thread executing MJOIN. It is allowed to continue and all other threads die. The MJOIN instruction generates continuation token <FP.IP + 1> for the last thread and thus execution is synchronized.

As code is systematically compiled from dataflow graphs and processor is multithreaded, instructions from unrelated threads will not compete for the same location of OM. There will always be an adequate number of MJOINs to prevent races between normal instructions. The exception is that there can still be a race between two or more MJOIN instructions competing for the same location. Each MJOIN instruction reads OM location, tests it, and writes it back, this must be atomic. To handle such a race condition, MJOIN is executed in exclusion. This is done in the following manner.

If the IF unit fetches MJOIN instruction and MJOIN lock is set then the instruction is not passed to OFU else IFU sets the global MJOIN lock and passes MJOIN instruction to the OFU. The MJOIN lock thus prevents any other TRSs to execute MJOIN as next instruction. As all units operate asynchronously with handshake signals other TRS pipelines are stalled whenever they fetch MJOIN instruction. However the pending instructions in the pipeline can still continue to execute. After MJOIN is being executed

atomically, RS unit unsets the MJOIN lock line after updating the OM location.

4.4 Data tranfer to and from global memory

Twine RISC model assumes the shared memory address space accessed by all TRSs. This memory is conceived as an I-structure like global memory. Any memory reference arising in a Twine RISC processor is sent out as a split phase transaction to I-structure memory controller. To support this there are LOAD/STORE/RESM instructions with hardware support of message processor and data queue. When a memory reference(LOAD/LOADX) is issued, the thread suspends; upon completion of the operation a value is sent from memory, a RESM instruction is executed and the thread reinitiates.

4.4.1 LOAD/LOADX : Data transfer from global memory to operand memory

LOAD instruction takes two parameters. the first parameter specifies the OM location containing global memory address whereas the second parameter specifies the OM location where the data read is to be stored. In response to this instruction read request is sent to the memory controller of I - structure global memory through MP and thread is suspended. IFU picks up another thread from the token queue and continue execution: The Read request format is shown below.

<Read gma, ct, dr>

where

gma - address of global memory location

ct - continuation token

dr - destination register

When this Read request is satisfied by I-structure memory controller, it responds by reading content of location gma, say v, and sends a message

<Store v,ct,dr>

to the MP.

MP upon receiving return message inserts the values v,ct,dr in Data Queue and sends a continution token <FP.0> to Sequencer. The address 0 in CM stores RESM instruction.

Initially at power-on time OM is uninitialized and can be initialized through RS unit only. The LOADX instruction is used to initialize OM location.

Format of the instruction is LOADX a, x

Where

a - address of location in global memory (limited to 6 bits only)

In all respects LOADX is similar to LOAD instruction.

4.4.2 RESM : Complete data transfer and Resume

This is an extention of LOAD/LOADX instruction. The RESM instruction is stored in location 0 of CM. Upon completion of the memory read request a value is sent to Data Queue and a continuation token <FP.0> inserted in the Token Queue. When token with thread address 0 is picked up by any of the TRSs, RESM

instruction is executed. Upon execution of this instruction, a tuple <v,ct,dr> is read from Data Queue.

Value v is stored in register dr and thus data movement from global memory to OM is completed. Also new thread token <ct> is inserted in TQ so thread continue exactly from the location next to LOAD/LOADX instruction.

4.4.3 STORE/STOREX : Move data from operand memory to global memory

Write request is sent to the memory controller of I - structure global memory through MP and thread is continued. The Write request format is shown below

<Write value, gma>

value - data to be stored

gma - address of global memory location

Memory controller on receiving Write message stores value in location gma. If location gma has deferred list of pending LOADs then it sends messages

<Store value,CT,A>

to MP.

Where

CT - corresponding continuation token

A - corresponding destination register in OM

Similar to LOADX, STOREX is used to store data in a fixed block of first 64 words in I - structure memory. It takes two parame-

ters. The first parameter specifies the 6 bit address of location in global memory and the second parameter specifies the value to be stored in there.

4.5 Instruction Set Summary :

Instructions ADD,SUB,AND,OR,XOR,SFTL,SFTR,STORE,STOREX do not generate new thread token. The execution continues from the subsequent location. i.e IP <-- IP + 1. In other words, the thread continues.

MFORK generates upto 4 new thread tokens and the parent thread also continues.

For jump like instructions, the next location can not be determined by IFU till the execution is complete. Hence JMP,JZ,JP,JPZ,JNZ generate new thread token and the parent thread dies. i.e. IP is set to new thread fetched from TQ.

LOAD,LOADX generate new thread token with thread address 0 and consume one.

RESM generates new thread token and consumes one.

MJOIN may/may not generate new token and consumes one.

Thus once TQ and CM is loaded Twine RISC itself generates and consumes threads and extract parallelism(spatial) by allowing more than one TRSs to be active and executing different threads. With a very little compiler effort Twine RISC can execute threada in parallel.(See Appendix[illegible])

Chapter 5. Simulator and Performance Evaluation

5.1 Introduction and Overview

In this chapter, we discuss the simulator for Twine RISC. The simulator provides a useful tool for the development of its architecture and has been used to modify its original design. Various parameters of the architecture are dependent on the currently available VLSI technology which form the input to the simulator.

The rest of the chapter is organized as follows. In section 5.2 we discuss the structure of the simulator. Input and output interface of the simulator is also discussed in this section. Metrics for the performance and performance evaluation aspects are discussed in section 5.3. In section 5.4, we discuss the usefulness of this simulator and discuss how it had been used to enhance the design. Finally we conclude this chapter in section 5.5.

5.2 Simulator Structure

Simulation is divided into two parts viz; preparation of input data and execution.

5.2.1 Input Preparation

A program written in conventional language is first converted to its equivalent dataflow graph. This dataflow graph is then converted into machine language program of Twine RISC. Simulator requires 3 files viz; GMF, CMF, TQF as its input. Here GMF

file contains the initial global memory image. CMF file contains the program code in binary. TQF file contains the initial token queue image. These files are prepared as follows :

All immediate data are seperated from instructions as instructions only refer OM locations for operand values. These initial available data with other input data are put into file GMF(global memory file) at appropriate locations. These values moved to OM through LOADX instructions.

We provide Twine RISC instructions to the simulator. The code in current model of execution is fed manually. This can however be done through a compiler at a later stage. The instructions to the simulator are fed using mnemonics. This mnemonic instruction code is converted to machine level binary equivalent code by code converter. Binary coded instructions are stored in code memory image file CMF. A compiler can directly generate CMF file to use the simulator.

By looking at the CMF explicit threads are distiguished. Such thread addresses are kept in TQF(token queue file).

5.2.2 Execution

Simulator program(Main()) asks user to enter GMF, CMF and TQF file names. This can also be given as command line arguments to the simulator. Programs stores data from GMF into global I - structure memory simulated by it. It stores instructions from CMF into Code Memory and data(thread addresses) from TQF is inserted into Token Queue. After this initialization is done, Simulator enters the function Execute() with Do-While loop.

Initially all TRSs are inactive. They are given priority according to their tag no., i.e. TRS#1 has highest priority over others for fetching new thread address from TQ. Otherwise all TRSs simultaneously attempt to read TQ creates problem.

At simulated global clock tick #T, TRS#1.IFU() reads TQ and fetches instruction from CM. It also sets IP to IP + 1 if thread is strictly sequencial. Also it raises its status bit in active state(1). When TRS pipeline is empty this bit is turned to inactive state(0).

At clock tick #T+1, TRS#1.OFU() reads buffer, fetches operands from OM and writes data packet into buffer. At same instant, TRS#1.IFU() reads another instruction and writes data packet into buffer. Thus TRS#1 pipeline strats filling.

At the same time, if TQ is not empty(separate status bit is provided), TRS#2.IFU() reads thread address from TQ, fetches instruction from CM and start executing new thread. Thus under favourable conditions, after clock tick #T+4, all four TRSs are active, executing different threads of computation.

In case of just starting address is kept in TQ, TRS#1 continues execution of thread and generates new continution tokens which fill TQ.

Execute()-loop is terminated when following conditions are satisfied.

1. TQ is empty.
2. All TRSs are inactive.

Then program(Main()) asks user to enter global memory addresses where output data are stored and then displays computed

results and stores it into Result File.

Simulator organization and code structures are included in Appendix B.

5.3 Performance Metrics

Simulator is written on the base of architectural assumptions we made. Since it is difficult to time various units precisely we cannot compare performance of the Twine RISC architecture with other processors. Basically it is targeted for checking the performance of architecture in exploiting available parallelism as claimed previously.

Several program codes have been implemented in Twine RISC's native graph representations and run on the prototype simulator. The measured performances shows that Twine RISC indeed can execute dataflow graphs with its software environment which requires very little compiler efforts. Also in the codes tested on Twine RISC Simulator, the number of instructions required was nearly equal to that for conventional control flow processor. The Twine RISC processor's ability to exploit parallelism is evident when four diagnostic loops were run together and pipelines are kept full.

Sample programs and results are reported in Appendix C.

5.4 Some Design Issues

We had started with writing simulator on architectural and software support assumptions made in [12]. But it was quickly found that proposed architecture and its software environment

does not support each other.e.g. MFORK. That lead us to make changes both in software constructs and in architecture design. Initialization of OM came into picture when prototype simulator was ready. Which lead us to include LOADX instruction and also to maintain. To avoid a complex OM implementation, Data Queue inclusion was deemed right. Finally Token Queue management and explicit thread generation and synchronization requirement lead us to put RESM instruction at fixed location 0 in Code Memory and some change in instruction's format.

As simulator writing was in progress need for precise specifications arose. That lead us to develop suitable instruction set with its very careful coding to have minimum hardware at various stages to decode instructions. Also size of the various blocks are considered with available VLSI technology and state-of-the-art memory design.e.g. Operand Memory(Register File), Data Queue, Messege Processor, Buffers, Token Queue etc.

5.5 Summary

Simulator writing has provided us lot of feedback in making major changes in the architecture to make it foolproof. Performance evaluation shows that Twine RISC is able to fulfill its goals of executing dataflow graphs efficiently with economical architectural framework.

Chapter 6. Discussion, Conclusion and Future Work

6.1 Introduction and Overview

In this chapter, the basic philosophy that motivated our work is stated. The overall results of this work are summarized. To conclude, we suggest future research work needed to support our work.

6.2 Philosophy

Twine RISC processor design is targeted for enhancing the performance by exploiting both temporal and spatial parallelisms. Basic architectural framework was already there with essential software environment directives.[12]. Before going for its hardware implementation simulation was needed to detect design errors.

6.3 Summary

We have implemented the simulator for one Twine RISC Stream on SUN 3/60 under SUN OS using C. The simulator is based on event driven model and provides the time trace of a simulator run.

The instructions for Twine RISC simulator can be provided in mnemonics which are then converted to their binary equivalent using a code converter developed through this work.

Synchronization in pipeline is also observed through some test programs written in Twine RISC assembly language and run on

the simulator. Through the simulator runs it is clear that the architectural assumptions of Twine RISC exploiting maximum available parallelism is true.

6.4 Scope for Future Work

Simulator writing has provided us substantial feedback in making major changes in the design. The simulator however does not provide precise timing analysis, This can be incorporated for performance evaluation of the architecture.

Currently the input to the simulator is provided through hand coded mnemonics. A compiler interface can be developed for this purpose.

As Twine RISC processors can be used to build high performance parallel computer, an exact protocol to do so need to be developed.

There is a tremendous flexibility in hardware design that depends largely on available technology. This can also be explored, for example, one more write port to Operand Memory helps in its initialization which at present is done by a number of LOADX instructions.

Category	Instruction	Action
Airthmetic and Logic	ADD	integer add
	SUB	integer subtract
	AND	bitwise AND
	OR	bitwise OR
	XOR	bitwise XOR
	SFTL	shift left
	SFTR	shift right
Branch	JMP	direct jump
	JZ	jump on zero (equal to)
	JP	jump on positive (greater than)
	JPZ	jump on positive or zero (greater than or equal to)
	JNZ	jump on negative or zero (less than or equal to)
Data transfer from/to memory	LOAD	load from external memory to OM/register
	LOADX	" "
	RESM	move data from DQ to OM/register
	STORE	store to external memory from OM/register
	STOREX	" "
Thread gen. and synch.	MFORK	generation of multiple threads
	MJOIN	synchronization of multiple threads

Fig A.1-1. Instruction Set

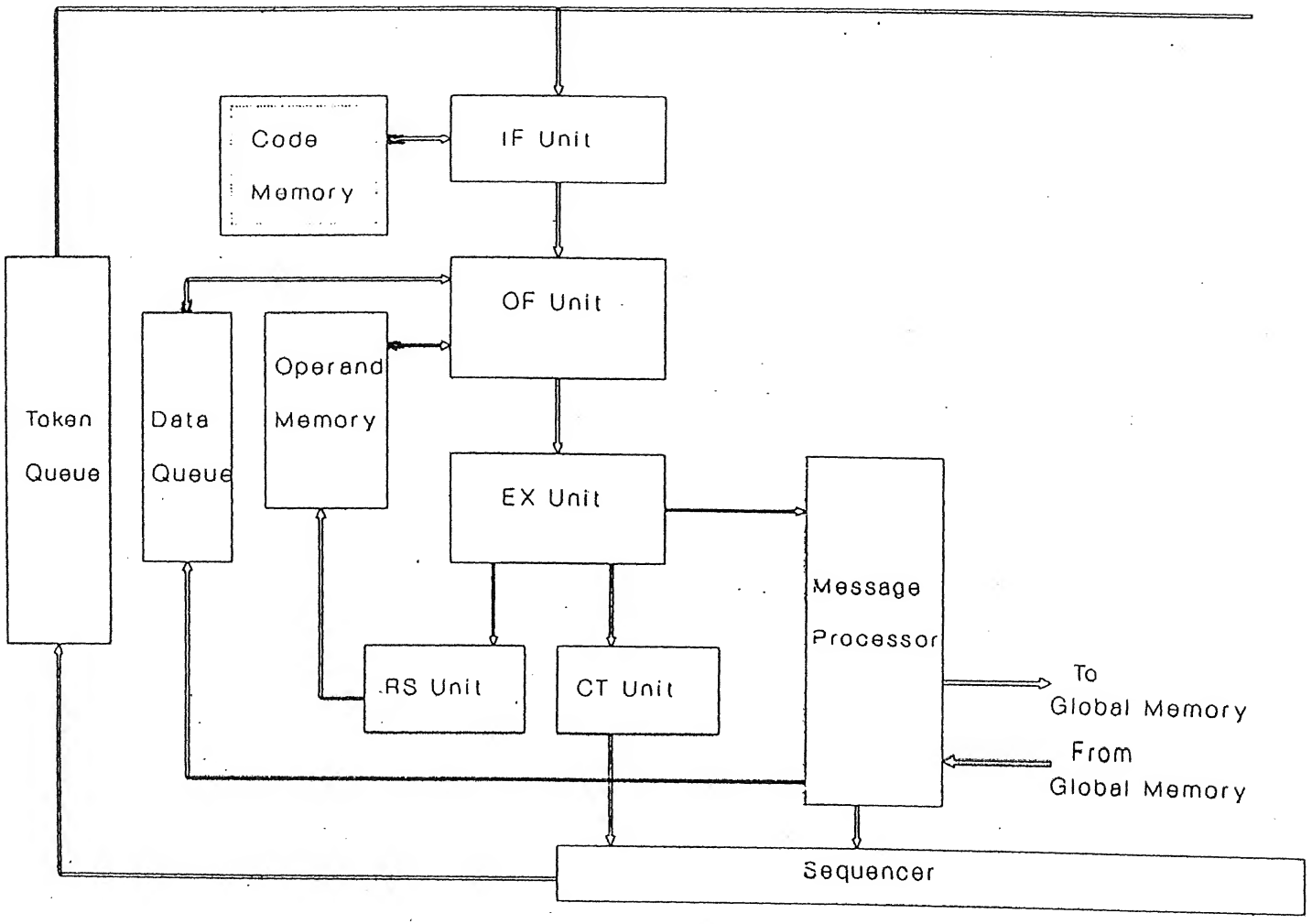


Fig.A.2-1. Twine RISC Processor Architecture

Appendix A : Instruction Set

This appendix consists of four sections. Section A.1 gives instruction set of Twine RISC. Section A.2 describes instruction flow in TRS pipeline. Section A.3 gives coding of instruction set. And finally A.4 gives the short summary of instruction set.

A.1 Instruction Set :

The instruction set of a Twine RISC processor is intended to be a simple extension of the RISC model. There are totally 19 different instructions, each of which is capable of being executed in a single clock cycle(Except MFORK instruction). Instructions are classified in four major categories viz. Arithmetic & Logic, Branch, Memory references and Generation and synchronization of multiple threads. (Fig A.1.1)

A.2 Instruction Execution in TRS pipeline :

A.2.1 Ordinary RISC like instructions :

a. ADD, SUB, AND, OR, XOR :

All these instructions fetch two operands from OM and store one result back to OM.

Syntax of these instructions is

opcode r1 r2 r3

r1 - left operand source register

r2 - right operand source register

r3 - destination register

e.g. consider opcode ADD

OFU fetches two operands [FP.r1] and [FP.r2], EXU adds them as [FP.r1] + [FP.r2] --> value and passes <value, r3> to RSU. RSU stores value in [FP.r3]. Execution continues from next location(i.e. IP + 1).

b. SFTL, SFTR :

As we are operating on 32 bit operands we can shift it by at most 32 bits. The shift count is stored in the instruction in 6 bits only. Thus we need to fetch only one operand from OM and store result back to OM.

Syntax of the instructions is

opcode a r2 r3

a - value specifies how many bits to be shifted(stored in instruction)

r1 - operand source register

r3 - destination register

e.g. consider SFTL a, r2, r3 instruction.

OFU fetches one operand [FP.r2], EXU operates and computes the result value as [FP.r2] << a --> value and passes <value, r3> to RSU. RSU stores value in [FP.r3]. Execution continues from IP + 1.

c. JMP

This instruction supports direct jump up to 18 bit range. As jump address is directly specified in 18 bits in the instruction, there is no operand fetch from OM.

Syntax of this instruction is

JMP x

x - 18 bit value specifies jump address

OFU does not fetch any thing from OM. x is treated as one operand. EXU generates continuation token <FP.x> and passes it to CTU. CTU forwards this continuation token to Sequencer which then inserts it into TQ. No result is written to OM.

d. JZ, JP, JPZ, JNZ

These instructions support conditional jump up to 12 bit offset range. As jump offset is directly specified in 12 bits in instruction, we need to fetch only one operand(in which condition value is stored) from OM.

Syntax of these instructions is

JCOND r1 x

r1 - condition operand source register

x - 12 bit value specifies jump offset

OFU fetches one operand [FP.r1], x is treated as another operand, EXU tests the condition [FP.r1] and if condition is true EXU generates continuation <FP.IP + x> else <FP.IP + 1> is generated. EXU passes this continuation token to CTU. CTU forwards this continuation token to Sequencer which then inserts it into TQ. No result is written to OM.

A.2.2 Special Instructions :

These instructions are extension of the RISC model.

a. MFORK

The MFORK instruction is a method of spawning parallel threads

of computation from within an executing thread.

New thread offsets are organized as n1,n2,n3,n4 each 8 bits. Which then grouped in a 32 bit number stored in OM. Thus only one fetch from OM is required.

Syntax of this instruction is

MFORK r2,r3

r2 - operand source register which contains a grouped number from which new thread offsets are derived.

r3 - destination register

OFU fetches one operand [FP.r2] from OM, EXU interprets [FP.r2] as four blocks of 8 bits each.

For each byte if byte value is nonzero then value is considered as an offset. EXU prepares a continuation token <FP.IP + offset value> and passes it to CTU. CTU forwards these continuation tokens to Sequencer.

One continuation <FP.IP + 1> is always there.

Number of new threads generated is derived as follows :

Bytes				No. of new threads
#1	#2	#3	#4	(value)
NZ	X	X	X	5
Z	NZ	X	X	4
Z	Z	NZ	X	3
Z	Z	Z	NZ	2

Here

Byte #1 is the most significant byte

NZ - nonzero value

Z - zero

X - don't care

EXU passes <value,r3> to RSU.

RSU stores value in [FP.r3].

b. MJOIN

The MJOIN instruction allows multiple threads to synchronize execution. Only one fetch from OM is required. This instruction decrements the content of the location specified by one and write back the result in the same location.

Syntax of this instruction is

MJOIN r2,r2

r2 - operand source register contains number of threads to be synchronized

r2 - destination register

OFU fetches one operand [FP.r2] from OM, EXU decrements [FP.r2] by 1 and tests it. If result value is zero then continuation <FP.IP + 1> is passed to CTU else thread dies. EXU also passes <result value,r2> to RSU. RSU stores value in register [FP.r2].

If continuation token <FP.IP + 1> is generated then it is forwarded to Sequencer by CTU.

c. LOAD, LOADX

These instructions are used to send a request to move data from global I - structure memory to OM.

Syntax of these instructions are

1. LOAD a x

a - operand source register contains address of global memory location

x - destination register

OFU fetches one operand [FP.a] from OM, EXU sends a read request to MP which then forwards it to memory controller of I -

structure memory and thread dies. The request format is

<Read [FP.a], FP.IP + 1, x>

[FP.a] - address of global memory location

FP.IP + 1 - continuation token

x - destination register in which data is to be moved

2. LOADX a x

a - 6 bit value specifies address of global memory location in instruction itself

x - destination register

OFU does not fetch any operand from OM, a is treated as an operand. EXU sends a read request to MP which then forwards it to memory controller of I - structure memory and thread dies. The request format is

<Read a, FP.IP + 1, x>

a - address of global memory location

FP.IP + 1 - continuation token

x - destination register in which data is to be moved

When this read message is processed by I - structure memory controller, it responds by reading contents of the location and sends a message to MP. The message format is

<Store v, FP.IP + 1, x>

v - data value

Upon receiving a return message from memory controller the MP inserts the values v, FP.IP + 1, x in Data Queue and sends a continuation token <FP.0> to Sequencer. Here continuation token <FP.0> corresponds to instruction RESM in CM.

d. RESM

When a memory operation LOAD/LOADX is issued, upon completion of the operation a value is sent to Data Queue, and a RESM instruction is executed to move data from Data Queue to OM and the thread reinitiates. This is the only instruction fetches operands from Data Queue.

Syntax of this instruction is

RESM

OFU fetches data

<v, FP.IP + 1, x> from Data Queue instead of OM.

v and FP.IP + 1 become two operands and x the destination register.

EXU passes <v,x> to RSU.

EXU also prepares a continuation token <FP.IP + 1> which is then forwarded to CTU.

RSU stores v in register [FP.x] and finally data is moved into OM.

CTU sends <FP.IP + 1> to Sequencer which then inserts it into TQ.

Here <FP.IP + 1> is not <FP.1> as RESM is at <FP.0> but this continuation token is read from Data Queue.

e. STORE, STOREX

These instructions are used to send a request to move data from OM to global I - structure memory.

Syntax of these instructions are

1. STORE x, a

x - operand source register contains address of global memory

location

a - operand source register from which data is to be moved to global memory

OFU fetches two operands [FP.x] and [FP.a] from OM, EXU sends a message to MP which is then forwarded to memory controller of I - structure global memory and thread continue. The message format is

<Write [FP.a], [FP.x]>

[FP.x] - address of global memory location

[FP.a] - value to be written

Memory controller upon receiving Write message stores value [FP.a] in location [FP.x]. If location [FP.x] has deferred list of pending LOADs then memory controller sends store messages to MP. The message format is

<Store v, CT, A>

v - data value

CT - corresponding continuation token

A - corresponding destination register

2. STOREX x, a

x - 6 bit value specifies address of global memory location in instruction itself

a - operand source register from which data is to be moved to global memory

OFU fetches one operand [FP.a] from OM, x is treated as other operand.

EXU sends a message to MP which is then forwarded to memory controller of I - structure global memory and thread continue.

The message format is

<Write [FP.a], x>

x - address of global memory location

[FP.a] - value to be written

Memory controller upon receiving Write message stores value [FP.a] in location x. If location [FP.x] has deferred list of pending LOADs then memory controller sends store messages to MP. The message format is

<Store v, CT, A>

v - data value

CT - corresponding continuation token

Execution continues from IP + 1.

A.3 Instruction Set Coding

The instruction set of Twine RISC is coded in such a way that by decoding minimum number of bits at various units of the pipeline instructions are recognized. There are 19 instructions in the instruction set sparsed over 6 bits of opcode.

IFU : IFU decodes the first bit of the opcode and decides the location of next instruction whether it comes for subsequent location, i.e IP + 1 or not. In the later case, new thread address is taken up from Token Queue for execution. If first bit is 0 then IP is incremented to IP + 1 else new thread address is fetched from TQ.

Also we need to identify MJOIN instruction at this stage only. This is needed for seting the global MJOIN lock and thus providing MJOIN execution in exclusion. Opcode for MJOIN is 111111 (all

1's) which can be decoded with minimum hardware support.

OFU : Here we need to recognize whether OFU has to fetch two operands, one operand or no operand from OM. Also it identifies the RESM instruction for which the operands are fetched from Data Queue. By decoding last two bits of opcode this can be done as

00 - two fetches from OM

01 - fetch from DQ

10 - no fetch from OM

11 - one fetch from OM

In some instructions, second operand is specified in the instruction itself. This is implemented by decoding yet another bit.

a. last 12 bits are treated as second operand or not, i.e. for conditional jumps and other instructions in one fetch category like SFTL,SFTR,MFORK etc.

b. last 18 bits are treated as an operand or not, i.e. for JMP and LOADX in no fetch category.

EXU : By looking at first 5 bits all instructions can be decoded.

Coding of instruction set is given in table#. Fig. A-3-1

Summary of instruction set is given in table#. Fig. A-3-2.

MSB	00 (2 fetch)	01 (DQ fetch)	10 (no fetch)	11 (1 fetch)
0000	ADD			
0001	SUB			
0010				
0011				
0100	AND			SFTL
0101	OR			SFTR
0110	XOR			MFORK
0111	STORE			STOREX
1000				JZ
1001				JP
1010				JPZ
1011				JNZ
1100			JMP	
1101			LOADX	
1110				LOAD
1111		RESM		MJOIN

Fig. A.3-1. Instruction Set Coding

Format	Frame Operations	Continuations	Outgoing Messages
ADD r1 r2 r3	[FP.r1] + [FP.r2] -> [FP.r3]	‹FP.IP+1›	-----
SUB r1 r2 r3	[FP.r1] - [FP.r2] -> [FP.r3]	‹FP.IP+1›	-----
AND r1 r2 r3	[FP.r1] & [FP.r2] -> [FP.r3]	‹FP.IP+1›	-----
OR r1 r2 r3	[FP.r1] | [FP.r2] -> [FP.r3]	‹FP.IP+1›	-----
XOR r1 r2 r3	[FP.r1] ^ [FP.r2] -> [FP.r3]	‹FP.IP+1›	-----
SFTL a r2 r3	[FP.r2] << a -> [FP.r3]	‹FP.IP+1›	-----
SFTR a r2 r3	[FP.r2] >> a -> [FP.r3]	‹FP.IP+1›	-----
JMP x	-----	‹FP.x›	-----
JCOND r1 x	test [FP.r1]	‹FP.IP+x› or ‹FP.IP+1›	-----
LOAD a x	[FP.a]	---	‹Read [FP.a], FP.IP+1, x›
LOADX a x	-----	---	‹Read a, FP.IP+1, x›
STORE x a	[FP.x], [FP.a]	‹FP.IP+1›	‹Write [FP.a], [FP.x]›
STOREX x a	-----	‹FP.IP+1›	‹Write [FP.a], x›
RESM	-----	‹FP.IP+1›[*0]	-----
MFORK r2 r3	[FP.r2] v -> [FP.r3] [*1]	‹FP.IP+1›, [*2] ‹FP.IP+n1›, ‹FP.IP+n2›, ‹FP.IP+n3›, ‹FP.IP+n4›	-----
MJOIN r2 r2	[FP.r2] - 1 -> [FP.r2]	If [FP.r2] = 0 then ‹FP.IP+1› else none	-----

[*0] - ‹FP.IP+1› read from Data Queue

[*1] - v is number of new threads generated

[*2] - n1, n2, n3, n4 nonzero (see MFORK A.2.2 a)

Fig.A.3-2. Instruction Set Summary

Appendix B : Code Structure for Simulator

This appendix gives basic code structure for Simulator.

B.1 Structure of Simulator

```
main()
{
        input();
        initialization();
        do {
                execution();
           } while (condition#1);
        output();
}
input()
{
        read GMF,CMF,TQF;
}
initialization()
{
        load global memory;
        load code memory;
        load token queue;
        set status of TRSs = 0;
        set status of all buffers = 0;
}
```

```
execution()
{
        TRS#1(),TRS#2(),TRS#3(),TRS#4();
}
output()
{
       ask for gm addresses;
       display resluts;
       store results to RF;
}
TRS#1()
{
       read TQ;
       set TRS#1.status = 1;
       do {
                IFU();
                OFU();
                EXU();
                RSU(),CTU(),SEQ();
           } while(condition#2);
       set TRS#1.status = 0;
}
IFU()
{
       fetch instruction from CM;
       decode opcode;
       set IP;
```

```
        forward data to buffer#1;
}
OFU()
{
        read buffer#1;
        set buffer#1.status = 0;
        decode opcode;
        fetch operands;
        forward data to buffer#2;
}
EXU()
{
        read buffer#2;
        set buffer#2.status = 0;
        decode opcode;
        operate on data;
        forward result to buffer#3,
        forward continuation to buffer#4;
}
RSU()
{
        read buffer#3;
        set buffer#3.status = 0;
        send handshake to EXU;
        store data in OM;
}
CTU()
```

```
{
        read buffer#4;
        set buffer#4.status = 0;
        forward continuation to SEQ;
}
SEQ()
{
        insert continuations into TQ;
}
```

Appendix C : User's Mannual and Test Programs

This appendix consists of two sections. Section C.1 gives directives to run the Twine RISC simulator. Section C.2 shows how Twine RISC extracts parallelism from programs. Two simple example programs are considered. For simplicity multiplication instruction is included in the instruction set.

C.1 User's Mannual

Simulator is written using 3 files.

- [func.c] contains all functions used in main file.
- [a.c] is main file, calls functions.
- [global.h] contains global variables and data structures used in simulator.

Executable file [sim] is to be generated using Makefile specifically written for this simulator.

To run simulator give command

sim gmf cmf tqf rsf

Before running the simulator files gmf,cmf,tqf, and rsf are to be prepared.

[gmf] file contains initial global memory image. It contains global memory locations and data values.

[cmf] file contains code memory image. It contains CM locations and instructions. To prepare this file first one has to prepare temp file consists of locations, mnemonic instruction code. A code converter converts this temp file to machine level binary equivalent file. For that [convert] is written.

[tqf] file contains explicit thread addresses known before hand.

[rsf] file contains global memory locations from where result values can be stored after computation.

Simulator generates output file [rout], which contains global memory locations given in [rsf] and values computed in program.

All these image files are given for test program 1. (see fig#C.2.2,3,4)

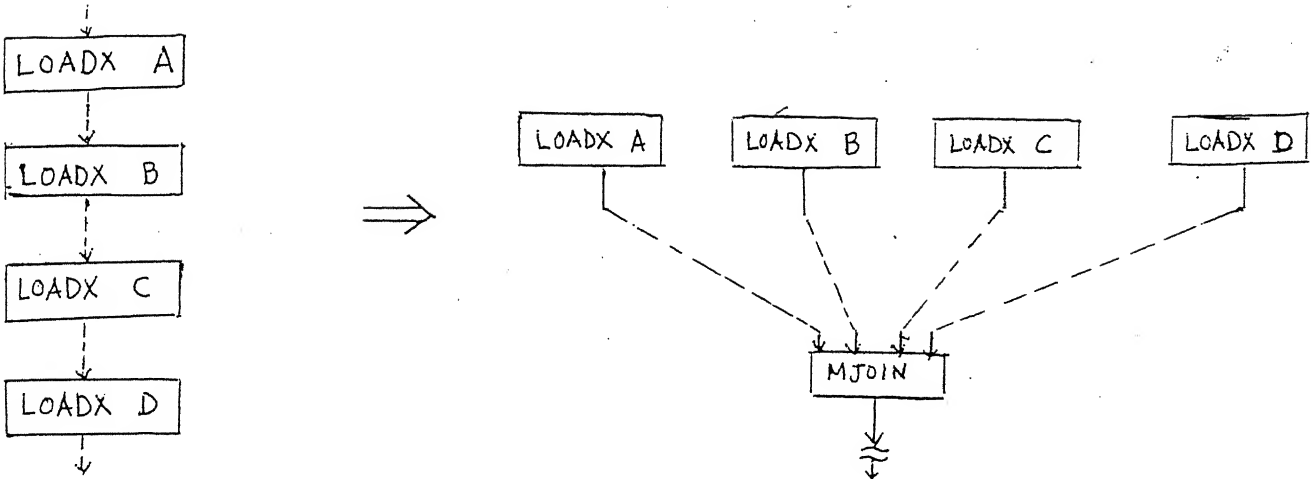
C.2 Test Programs

Program 1.

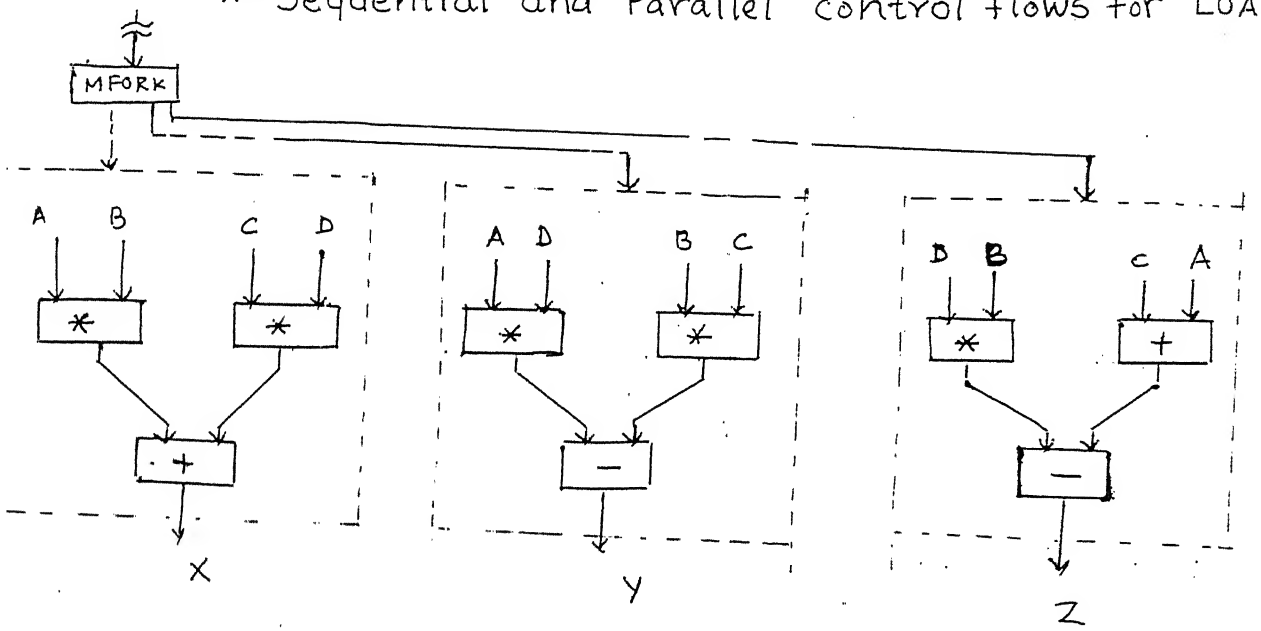
Compute X,Y,Z.

```
X = [A*B] + [C*D];
Y = [A*D] - [B*C];
Z = [D*C] - [C+A];
```

Here it is evident that once A,B,C, and D are available, X,Y, and Z can be computed concurrently. Also A,B,C,D can be moved to operand memory registers by dispatching concurrent LOADs. Parallel control flow for LOADs is done with the help of MJOIN instruction, which allows execution to continue only after A,B,C, and D are moved into OM. As these LOADs are explicitly known, their addresses are inserted in TQ before execution. X,Y,and Z is computed simultaneously with the help of MFORK instruction, which generates parallel threads of computation for X,Y, and Z. X,Y, and Z then can be synchronized for further computations. As this program does not have any iterations frame pointer FP is same(i.e.=0) for all instructions.



A. Sequential and Parallel control flows for LOADs.



B. X, Y, Z computed concurrently.

Fig. C.2-1.

```
location   instruction

    0      resm            /*resume instruction at location 0*/
    1      loadx 1 1       /*load mjoin register 1 (for synchronizing loadxs)*/
    2      jmp 12          /*jump to synch 1*/
    3      loadx 2 2       /*move A to OM register 2*/
    4      jmp 12          /*jump to synch 1*/
    5      loadx 3 3       /*move B to register 3*/
    6      jmp 12          /*jump to synch 1*/
    7      loadx 4 4       /*move C to register 4*/
    8      jmp 12          /*jump to synch 1*/
    9      loadx 5 5       /*move D to register 5*/
   10      jmp 12          /*jump to synch 1*/
   11      loadx 6 6       /*load mfork register 6 (for generating new threads)*/
   12      mjoin 1         /*synch 1*/
   13      mfork 6 7       /*new threads generated*/
   14      mul 2 3 10      /*thread #1 parent*/
   15      mul 4 5 11
   16      add 10 11 12
   17      jmp 25          /*jump to synch 2*/
   18      mul 5 2 13      /*thread #2*/
   19      mul 3 4 14
   20      sub 13 14 15
   21      jmp 25          /*jump to synch 2*/
   22      mul 5 3 16      /*thread #3*/
   23      add 4 2 17
   24      sub 16 17 18
   25      mjoin 7         /*synch 2*/
   26      storex 12 12    /*store X in global memory location#12*/
   27      storex 15 15    /*store Y in global memory location#15*/
   28      storex 18 18    /*store Z in global memory location#18*/


        cmf : code memory image file (mnemonic code) for prog.1
```

Fig. C.2.2

location	data value	
1	6	/*mjoin*/
2	10	/*A*/
3	5	/*B*/
4	6	/*C*/
5	2	/*D*/
6	2309	/*mfork #n1=5, #n2=9*/

A. gmf : global memory image file

FP.IP
0 . 1
0 . 3
0 . 5
0 . 7
0 . 9
0 . 11

B. tqf : initial token queue image

location
12
15
18

C. rsf : result locations file

location	data value	
12	62	/*X=(A*B)+(C*D)*/
15	-10	/*Y=(A*D)-(B*C)*/
18	-6	/*Z=(D*B)-(C+A)*/

D. rout : result output file

Files for program 1

Program 2.

Compute vector inner product.

```
for i=1 to n
S = S + A[i]*B[i];
```

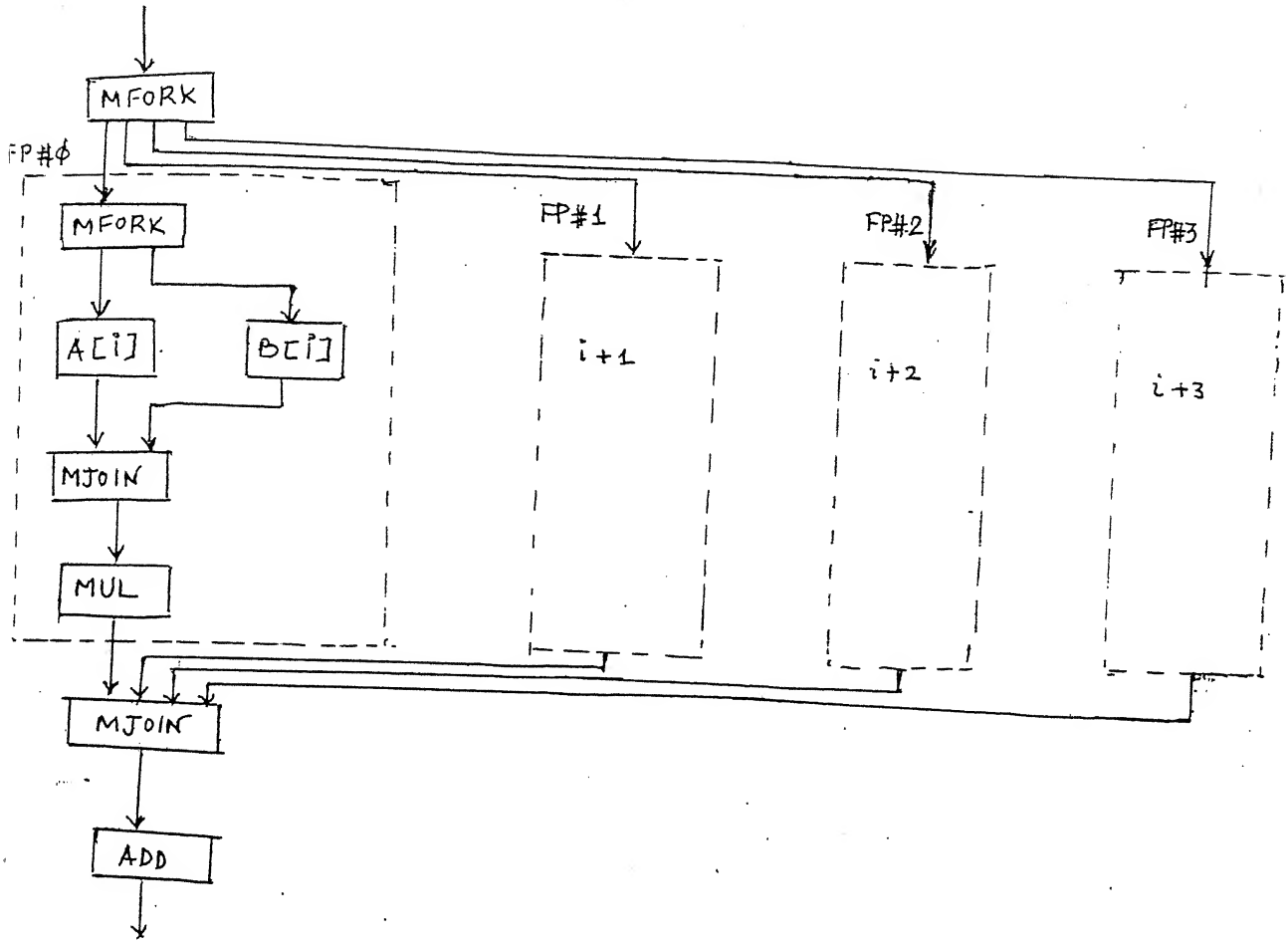
Here parallelism is exetracted in two ways.

1. A[i] and B[i] is simultaneously loaded.

2. For different i, A[i]*B[i] computed concurrently and then are added.

To compute all iteration codes in parallel, we have to provide different frame pointers FP for each i. So that with same code block in instruction memory execution is performed on different operand register sets.(see fig# C.2.4)

However, the programs stand "apparently" sequential, much parallelism is exploited by Twine RISC architecture -- all index calculations, loads and multiplications can be done in parallel.



Concurrent loads and iterations.

A[i] ⇒

ADD A i Ai

LOAD Ai Aix

Fig. C.2.4.

Netherlands, Springler-Verlag LNCS 259, June 1987.

[8] M. Beck, R. Johnson, and K. Pingali. From Control Flow to Dataflow. Journal of Parallel and Distributed Computing 12, 1991. Pages 118-129.

[9] D. Culler, G. Papadopoulos. The Explicit Token Store. CSG Memo 312, MIT Lab for Computer Science, June 1990.

[10] K. Hwang, F. Briggs. Computer Architecture and Parallel Processing. McGraw Hill, Computer Science Series, Int. Edition, 1985.

[11] R. Iannucci. Toward a Dataflow/von Neumann Hybrid Architecture. In Proc. 15th Int. Symp. on Computer Architecture, Honolulu, Hawaii, June 1988. Pages 131-140.

[12] R. Moona, S. Nandy, V. Rajaraman. Twine RISC : An Architecture for Simultaneous Execution of Multiple Threads. [Personal Communications].

[13] R. Nikhil, Arvind. Can Dataflow Subsume von Neumann Computing? In Proc. 16th Int. Symp. on Computer Architecture, Jerusalem, Israel, June 1989. Pages 262-272.

[14] G. Papadopoulos, D. Culler. Monsoon : an Explicit Token Store Architecture. In Proc. 17th Int. Symp. on Computer Architecture, Seattle, Washington, May 1990. Pages 82-91.

[15] G. Papadopoulos, K. Traub. Multithreading : A Revisionist View of Dataflow Architectures. In Proc. 18th Int. Symp. on

Computer Architecture, Toronto, Canada, May 1991. Pages 342-351.

[16] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In Proc. 16th Int. Symp. on Computer Architecture, Jerusalem, Israel, June 1989. Pages 46-53.

[17] I. Watson, J. Gurd. A Practical dataflow Computer. Computer 15(2), 1982. Pages 51-57.

[18] T. Yuba, T. Shimada, K. Hiraki, and H. Kashiwagi. SIGMA - 1 : A Dataflow Computer for Scientific Computation. Technical Report, Electrotechnical Laboratory, Japan, 1984.